AD-A085 515 VIRGINIA POLYTECHNIC INST AND STATE UNIV WASHINGTON --ETC F/G 9/2
A SIMULA PROGRAM FOR SLR(1) PARSING.(U)
JAN 80 J J MARTIN AFOSR-79-0021
UNCLASSIFIED VPI/SU-TM-80-1 AFOSR-TR-80-0445 NL

1 OF 1
AD A085 515

END
DATE FILMED
7-80
DTIC

EXTENSION DIVISION

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE
GRADUATE PROGRAM IN NORTHERN VIRGINIA

P. O. Box 17186
Washington, D. C. 20041
(703) 471-4600

A SIMULA* PROGRAM FOR SLR(1) PARSING†

Johannes J. Martin

Technical Memorandum No. 80-1

January 1980

ABSTRACT

The report describes a SIMULA program that parses character strings according to a grammar preprocessed by a parser generator [1]. The program does not furnish error recovery. This report provides all information necessary to use the program including a detailed description of the format of the parse tree produced and the symbol table employed.

Keywords and Phrases: SLR(1) parser, SIMULA

CR Categories: 4.12, 5.23

---

* SIMULA is a registered trademark of the Norwegian Computing Center, Oslo, Norway.

† Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, under Grant No. AFOSR-79-0021. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

(AFSC)

AI
NO nd is
Th -12 (7b).

A. officer

Preface

This report is addressed to those potential users of a context free parser program who have a good background in the theory of parsing. Also some elementary knowledge of the programming language SIMULA is needed in order to understand the interface required.

The report is meant to supply an expert programmer with all information necessary for using the program and for assessing its performance. It is not meant, however, to motivate the reader to use this parser rather than some other parser program.

A second report (by Richard J. Orgass) that gives more background information, addresses some of the finer points of the work and, thus, promotes its use may be forthcoming.

1. General Properties of the Parser.

The parser described in this report is an SLR(1) device. Originally, it was written by F.C. Druseikis as part of the interactive program verification system devised by D.E. Britton (1977). The version descibed here has been extracted from the verification program and enhanced by an improved file manipulation system (DIALOG, Orgass,R.J.,1979). The program produces a parse tree and an (optional) source listing from a given program or program segment. Parsing is performed according to a parse table which must have the format defined in [1] and hence may be automatically produced by the parser generator described by the same reference. (Aho and Ullman [2] is an excellent reference on the theory of SLR parsing and the construction of actual parsers).

The parser program is written as a SIMULA CLASS named slr_parser. The procedure (part of the CLASS slr_parser) which does the actual parsing is called 'parse'.

Suppose, the file 'parsetbl' contains a parse table produced by the parser generator mentioned. Then, the parser is initialized by

```
REF(slr_parser) parser; REF(Infile) parsetbl;

parser :- NEW slr_parser(parsetbl);
```

The parsing action itself is envoked by the function reference

```
tree:- parse(input, root);
```

Here 'input' is the program or program segment which is supposed to be parsed. This program must first be packaged into to an object of CLASS 'stream'. The purpose of this packaging step is to allow both character strings (TEXT objects) as well as files (Infile objects) to be used as input.

For the above example, this packaging could be done by either of the following statements:

(i)
```
input:-NEW stream(program_text, FALSE);
```

Here we assume that 'program_text' is a TEXT object; therefore, the second parameter, which indicates whether or not a file is used, is set to FALSE.

(ii)
```
input:-NEW stream("PROGRM PLI", TRUE);
```

"PROGRM PLI" is the file name and file type of of a file that contains the program to be parsed; consequently, the second parameter is set to TRUE.

The second parameter of 'parse' ('root' in the example above) is a TEXT object whose value is the name of the non terminal symbol which is to be used as the starting symbol (root node) of the parse tree. This name must be enclosed in angular brackets.

The value of 'parse' is an object of the CLASS 'parse_tree'. The description of the format of this CLASS and its constituent operations will make up most of the balance of this report.

The following program segment gives a complete example of the use of 'slr_parser' and 'parse' with all necessary declarations.

```
REF(Infile) parse_tbl;
REF(slr_parser) parser;
REF(stream) input;
REF(parse_tree) tree;

  parser :- NEW slr_parser(parse_tbl);

  input  :- NEW stream("PROG PL4", TRUE);

  tree   :- parse(input, "<pgm>");
```

If a source listing is to be produced, a file must be declared and opened for receiving the listing. A special file type, called a videofile, has been defined as part of the parser. This file type provides both a SIMULA Outfile and an optional terminal listing. The terminal listing can be supressed by setting the variable 'blackout' equal to TRUE.

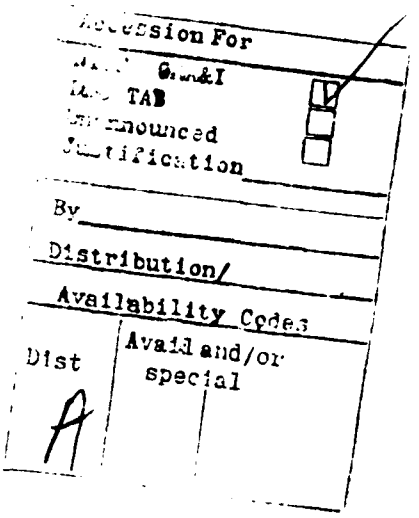
So, in order to obtain a source listing a videofile with the name 'sourcelist' must be created. The following sequence of code can be used to accomplish this.

```
REF(videofile) sourcelist;

   sourcelist :- NEW videofile("MYPROG");

    COMMENT: 'MYPROG' will be the file name of the
             new file, the file type, since not given,
             will be 'log' by default;

   sourcelist.blackout := FALSE;
   sourcelist.Open(Banks(80));
    COMMENT: if LRECL = 80 is wanted;
    .
    .
    .
       {rest of code for parsing etc.}
    .
  sourcelist.Close;
```



2. The Format of the Parse Tree.

2.1 Basic Information held in the Nodes of the Tree.

The parse tree produced is an object of the SIMULA CLASS 'parse_tree'. This CLASS has several subclasses, each one dealing with different type of a node of the tree.

These node types are

1) Nodes representing non-terminals (CLASS vnon),
2) " identifiers (CLASS vidn),
3) " numbers (CLASS vnum),
4) " keywords and other special symbols (e.g. ->) (CLASS vsym).

All of these nodes contain the print representation of the items they represent. For example '<formula>' or '<term>' would be used in a 'vnon' node if 'formula' or 'term' were names of nonterminals, 'begin', 'while' or ':=' may occur in a 'vsym' node, '12' in a 'vnum' node and 'my_name' in a 'vidn' node.

This information is provided so that a tree can be printed with readably labelled nodes.

In addition, nodes contain integer numbers, codes that refer to symbols and productions. The node type 'vnon' contains three such numbers, vnum two, vidn and vsym one.

The numbers in 'vnon' nodes have the following meaning: one number refers to the non-terminal stored (lhs), one refers to the right hand side of the production rule used (rhs) and the third (size) equals the number of symbols occurring in the right hand side of the production.

One of the numbers stored in a 'vnum' node is simply the converted number represented by the node, the other is the code for the symbol 'number'.

The numbers used in vsym nodes are codes for the respective symbols, the number in a vidn node is the code for 'ident'. In addition, vidn nodes contain pointers to the symbol table.

2.2 The Derivation of the Integer Codes for Symbols and Productions.

The codes that refer to symbols and productions are derived from the order in which these symbols and productions occur in the grammar given to the parser generator. Thus, in order to describe precisely the method by which these codes are constructed the input format for grammars as given to the parser generator must be explained.

Consider the grammar

```
<pgm> -> <stmt_list>.
<stmt_list> -> <stmt> | stmt_list>;<stmt>
<stmt>      -> <term> := <sum> | CALL ident
<sum>       -> <sum> + ident  | ident
<term>      -> ident | ident (<sum>)
```

This grammar is given to the parser generator as follows:

```
pgm         stmt_list .
stmt_list   stmt
            stmt_list ; stmt
stmt        term := sum
            CALL ident
sum         sum + ident
            ident
term        ident
            ident ( sum )
```

In short, a grammar is written one production per line; the left hand side non-terminal starts in column 1; if a nonterminal derives several right hand sides, then all these productions are listed in consecutive lines and only the first production shows the left hand non-terminal; the angular brackets around non-terminals are left out.

The parser needs, what is termed an augmented grammar; this is derived from the actual grammar by adding the production
<S´> -> |- w -| where ´w´ is a place holder for the actual starting symbol supplied with the call to ´parse´ (see invocation of the program ´parse´ above).

In order to simplify the following description of the assignment of integer codes to grammar symbols, it is convenient to assume the the parser generator accounts for the augmentation by prefixing the given grammar input with the line S´ |- -| with the new terminal symbols ´|-´ and ´-|´, which serve as left and right sentence delimiters.

Note: The symbols ´|-´ and ´-|´ are not used to physically delimit the input sentence. Rather, they are generated by the lexical scanner as signals to the parser indicating the beginning and the end of the input file.

2.21 Codes for Symbols.

From the augmented representation of the grammar, the integer codes for the symbols are derived by
1) scanning the grammar left to right and top to bottom,
2) recording symbols in the order in which they are found, non-terminals and terminals in seperate lists,
3) concatenating the lists when scanning is completed,
4) numbering the list elements consecutively starting with 1.
5) The numbers so assigned are the integer codes applied.

For the sample grammar, the final list has the following appearance:

```
 1     S
 2     pgm
 3     stmt_list
 4     stmt
 5     term
 6     sum
 7     |-
 8     -|
 9     .
10     ;
11     :=
12     CALL
13     ident
14     +
```

2.22 Codes for the Right Hand Sides.

The integer numbers assigned to the right hand sides of production rules are simply the ordinal numbers of the alternatives for a left hand side non-terminal.

For example, in the above grammar

'term := sum' would be called 1 whereas

'CALL ident' would be called 2.

2.3 The Format of the Nodes of the Parse Tree.

The actual format of the nodes is best explained by giving the skeleton of the SIMULA CLASS definitions for 'parse_tree' and its subclasses:

```
CLASS parse_tree(print_rep); TEXT print_rep;
 END of CLASS parse_tree;

parse_tree CLASS vnon(lhs,rhs,size); INTEGER lhs, rhs, size;
 BEGIN REF(parse_tree) ARRAY sons(1:size);
 END of vnon;

parse_tree CLASS vidn(ste,token); REF(symbol) ste;
                INTEGER token;
 END of vidn;

parse_tree CLASS vnum(val,token); INTEGER val, token;
  END of vnum;

parse_tree CLASS vsym(token); INTEGER token;
  END of vsym;
```

Summary of the fields of the different classes:

1) In the superclass 'parse_tree':
   'print_rep' is the print representation of the symbol.

2) In the class 'vnon' (non-terminal):
   'lhs' is the code of the non-terminal itself,
   'rhs' is the ordinal number of the alternative
   right hand side used,
   'size' is the number of symbols in the string
   that constitutes the right hand side.
   Not before mentioned:
   'ARRAY sons(1:size)' contains the roots of the
   subtrees to the nonterminal represented.

3) In the class 'vnum' (number):
   'val' is the converted (INTEGER) version of the
   numeral represented,
   'token' is the code for the terminal 'number'.

4) In the class 'vidn' (identifier):
   'token' is the code of 'ident',
   'ste' (symbol table entry) is the pointer
   to the corresponding symbol table entry
   (the format of such an entry is described below).

5) In the class 'vsym' :
'token' is the code of the keyword or operator represented.

2.4 Procedures Provided for Parse Trees.

The following procedures are part of the CLASS parse_tree and of all of the subclasses:

```
REF(parse_tree) PROCEDURE copy;
```

produces an independent copy of the given tree.

```
BOOLEAN PROCEDURE equals(x); REF(parse_tree) x;
```

compares the given tree with the tree x supplied as the parameter.

```
PROCEDURE ptre(pos); INTEGER pos;
```

prints an indented listing of the tree;
'pos' specifies the column in which printing is to start.
The different nodes are printed by the following formats:

```
vnon:  "vnon" print_rep lhs rhs size
vnum:  "vnum" print_rep val token
vidn:  "vidn" print_rep token
         symbol  sym  val        (see description of
                                  symbol table entry below)
vsym:  "vsym" print_rep token.
```

3. The Symbol Table.

The symbol table is programmed as a tree dictionary; the nodes of the tree are objects of the CLASS symbol:

```
CLASS symbol(sym,val); TEXT sym; INTEGER val;
  BEGIN REF(symbol) left, right; END of symbol;
```

The field 'val' may be used for storing semantic information after the parse is completed. Replacing the definition of 'symbol' by one that has more fields is, of course, possible as long as fields 'sym' and 'val' remain present.

4. Error Handling.

The parser has originally been written for an experimental interactive program verification system. Therefore, no error recovery mechanism has been provided. Upon a syntax error, the error is descibed by

1. giving the character position of the offending symbol,
2. printing the offending symbol and
3. printing a list of symbols acceptable in the given position.

Execution then terminates; a NONE parse tree is returned.

The program is terminated in a procedure called 'enterdebug' which, so far, does not do anything but write a short message and stop. This program is the logical hook for any future error recovery routine.

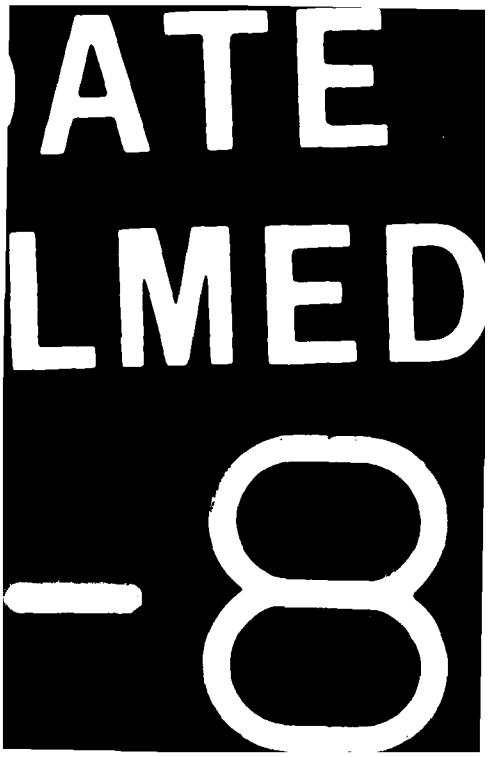
5. Availability.

The program is currently available at Virginia Tech's CMS (Conversational Monitor System running on an IBM 370/158) on the minidisk CSDULLES 191 under the name SLR SIMULA. The precise rules for encorporating this parser into another program as well as any later updates to this report will be found on the same disk under the name SLR MANUAL.

References

1. Druseikis, F.C.
   "SLR(1) Parser Generator, User's Manual,"
   technical report, Dept. of Comp.Sci.,Univ. of Arizona,
   March 1976.

2. Aho, A.V. and Ullman, J.D.
   " Principles of compiler design," Addison-Wesley 1977.


ATE
LMED
-8